Hash Tables

Mohsin Abbas

This topic is a part of your final exams

Part 1

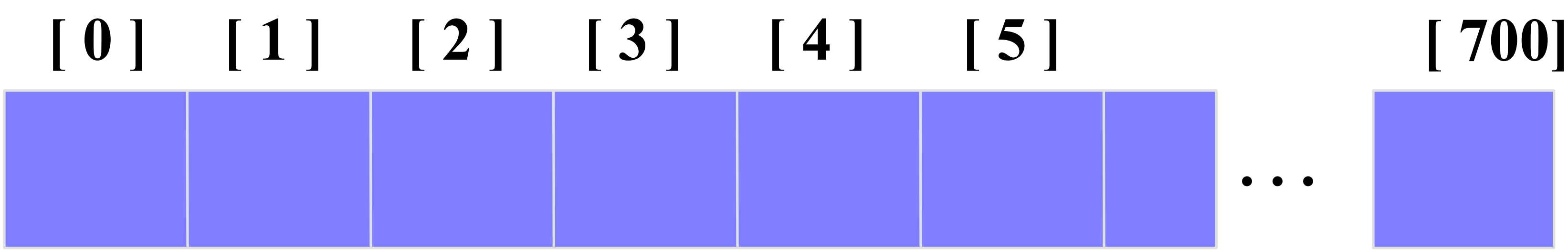
Introduction to Hash Tables

Introduction

- Hash tables store a collection of records with **keys**.
- There can be data associated with these keys called **mapped value**.
- The location (index) of a record depends on the **hash value** of the record's key.
- The hash-value (index location) is calculated based on HASH FUNCTIONS

What is a Hash Table ?

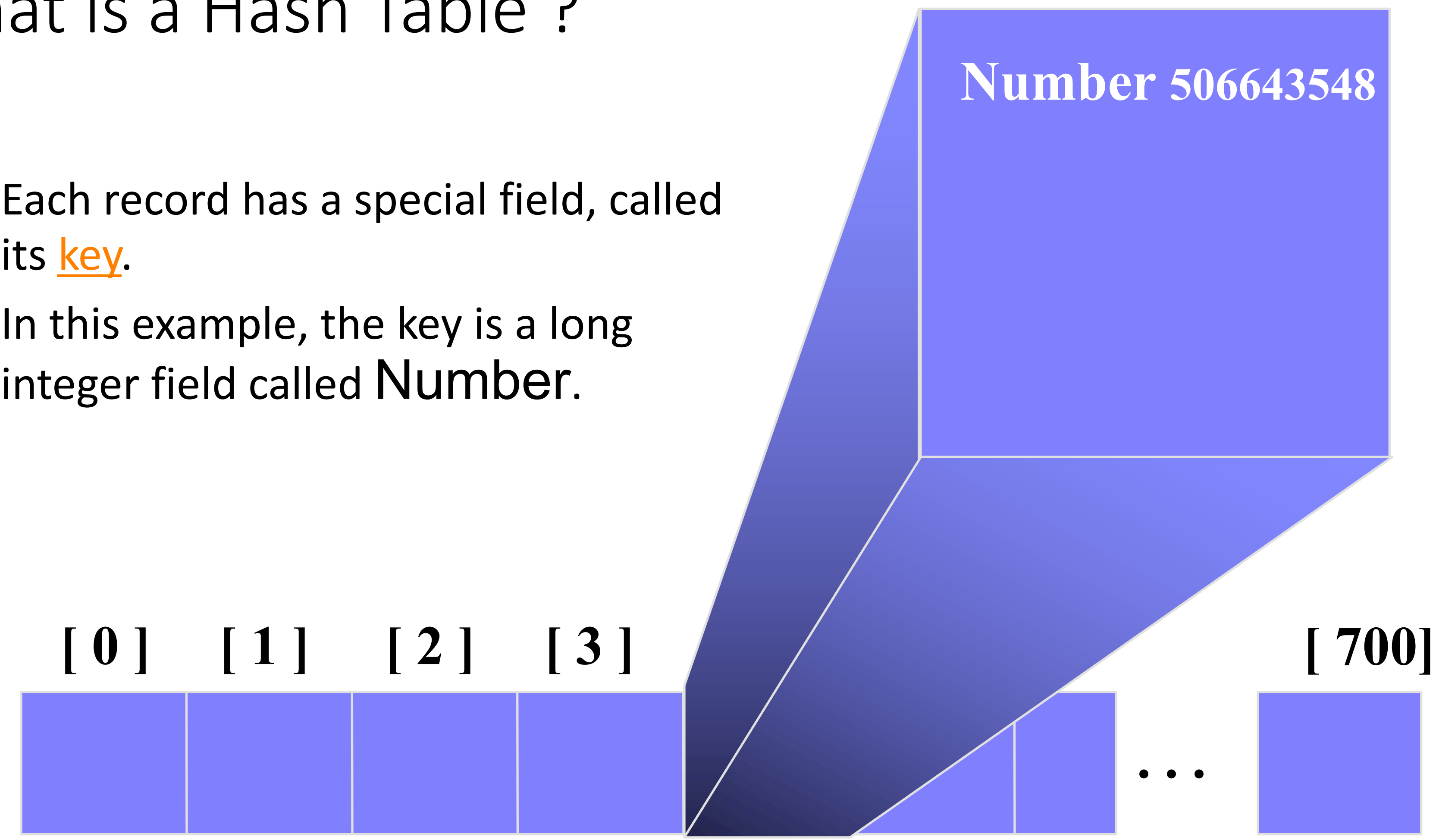
- The simplest kind of hash table is an array of records.
- This example has 701 records.
- Hash function is in our example is:
  - **`MOD size`**



An array of records

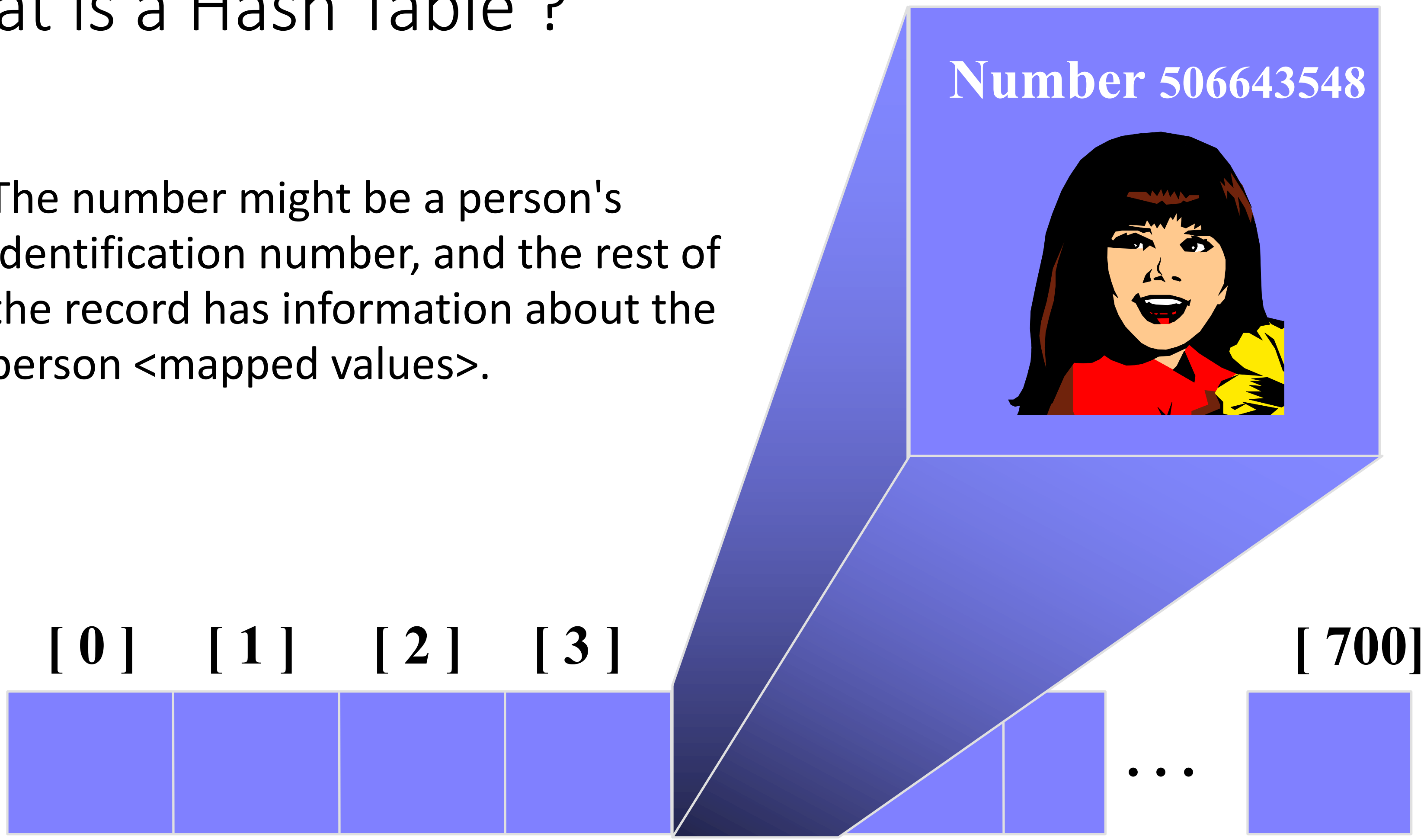
What is a Hash Table ?

- Each record has a special field, called its key.
- In this example, the key is a long integer field called Number.



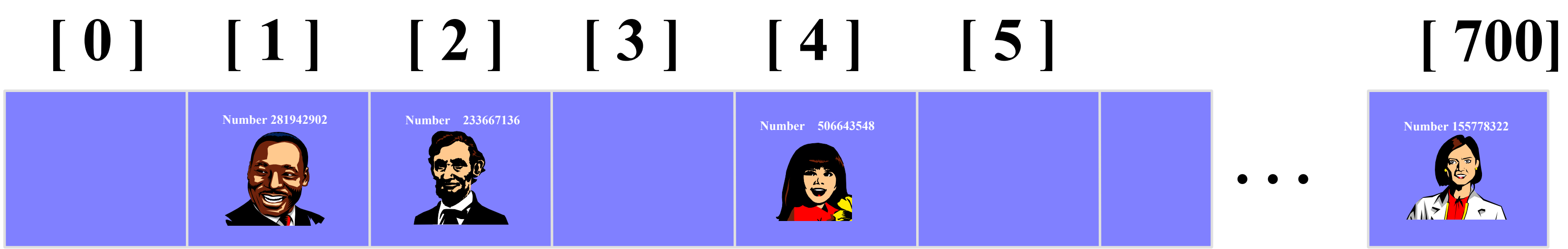
What is a Hash Table ?

- The number might be a person's identification number, and the rest of the record has information about the person <mapped values>.



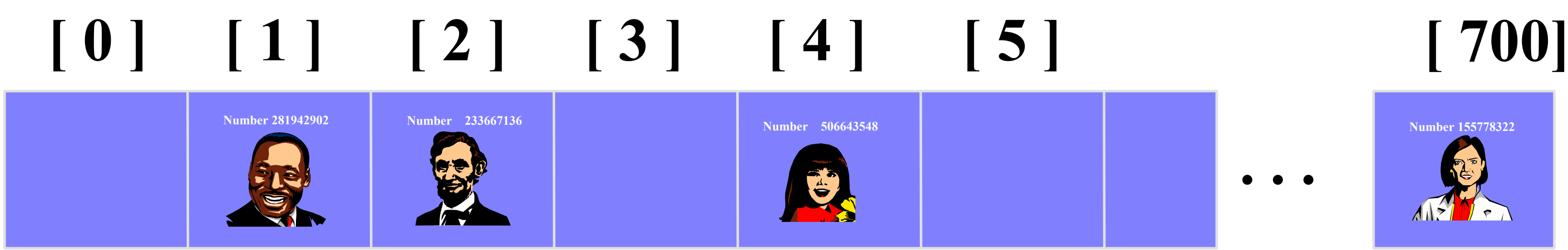
What is a Hash Table ?

- When a hash table is in use, some spots contain valid records, and other spots are "empty".



Inserting a New Record

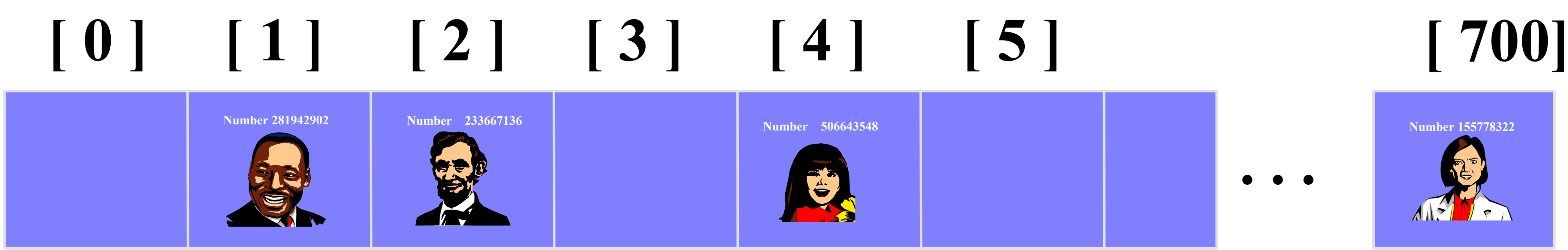
- In order to insert a new record, the **key** must somehow be **converted to** an array **index**
- Index is found using a ***HASH FUNCTION***
- The index is called the **hash value** of the key.



Inserting a New Record

- Typical way to create a hash value:

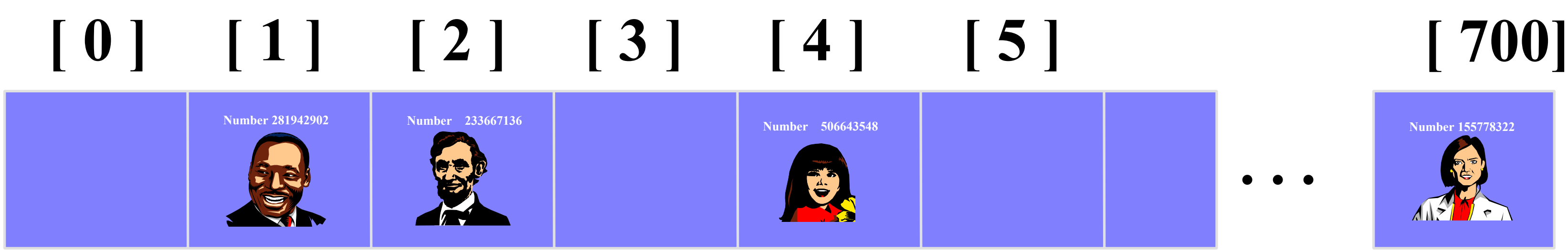
What is (580625685 % 701) ?



Inserting a New Record

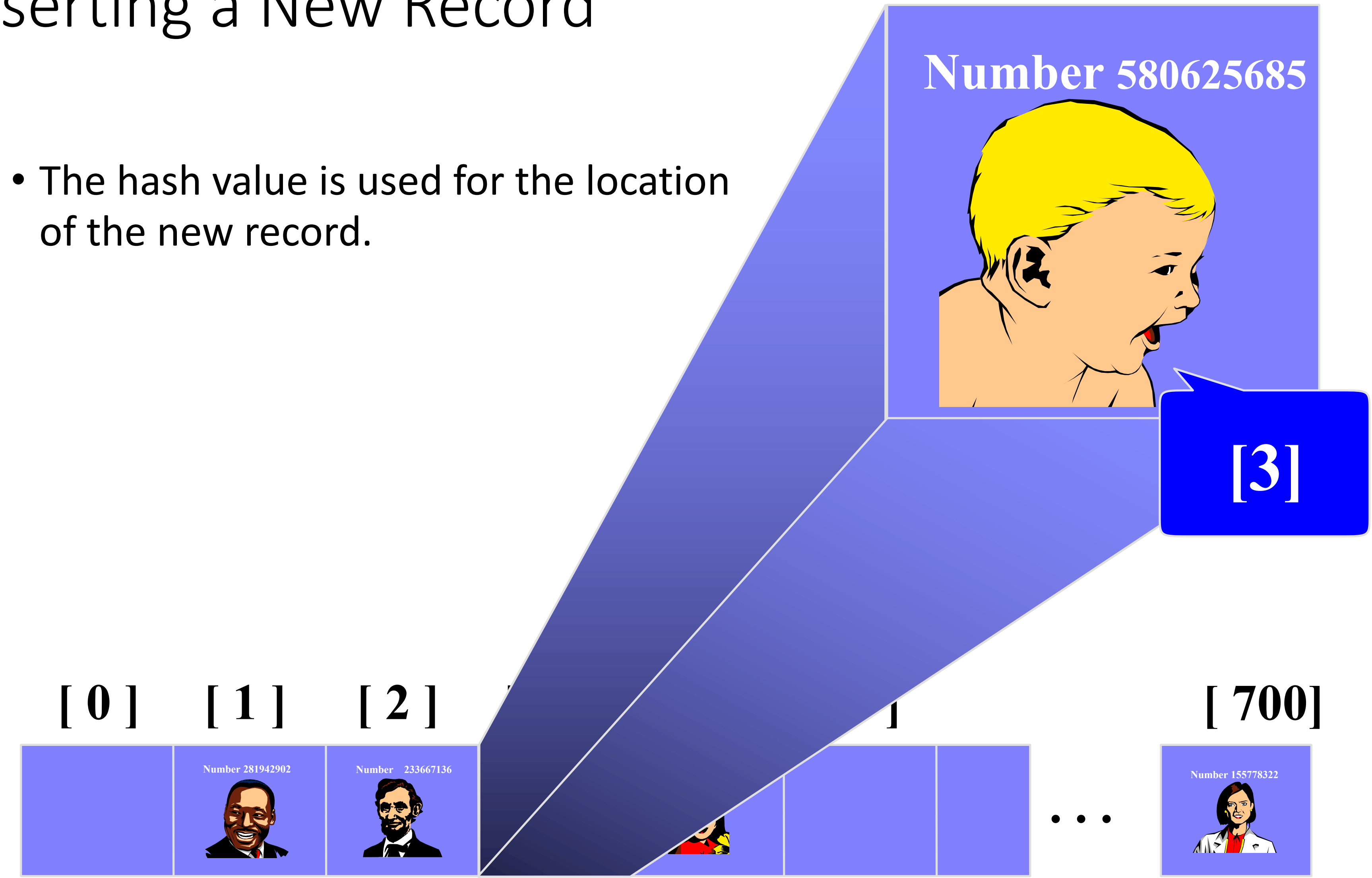
- Typical way to create a hash value:

What is (580625685 % 701) ?



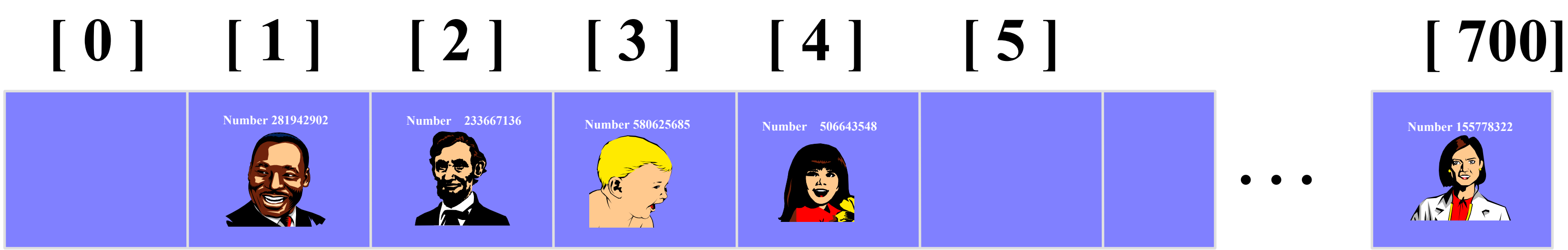
Inserting a New Record

- The hash value is used for the location of the new record.



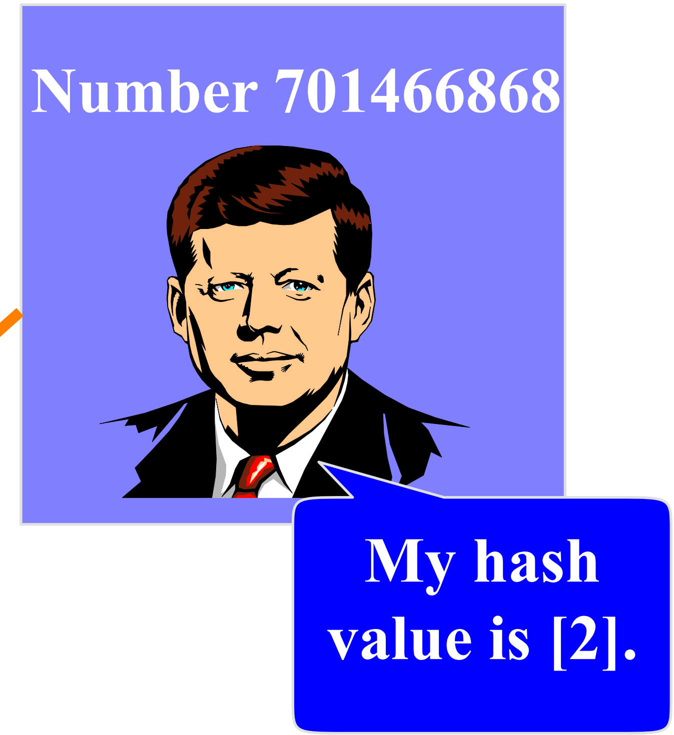
Inserting a New Record

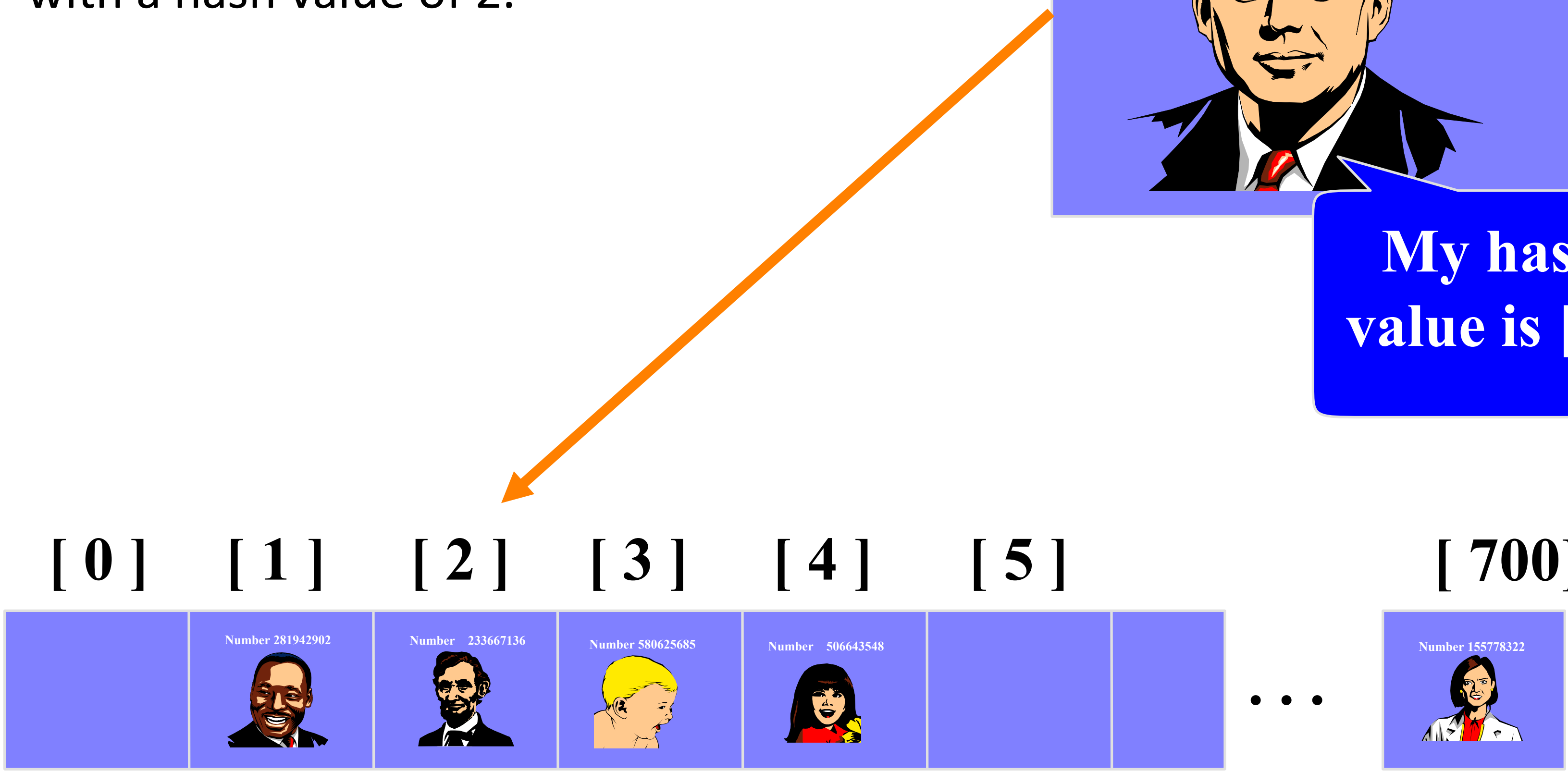
- The hash value is used for the location of the new record.



Collisions

- Here is another new record to insert, with a hash value of 2.

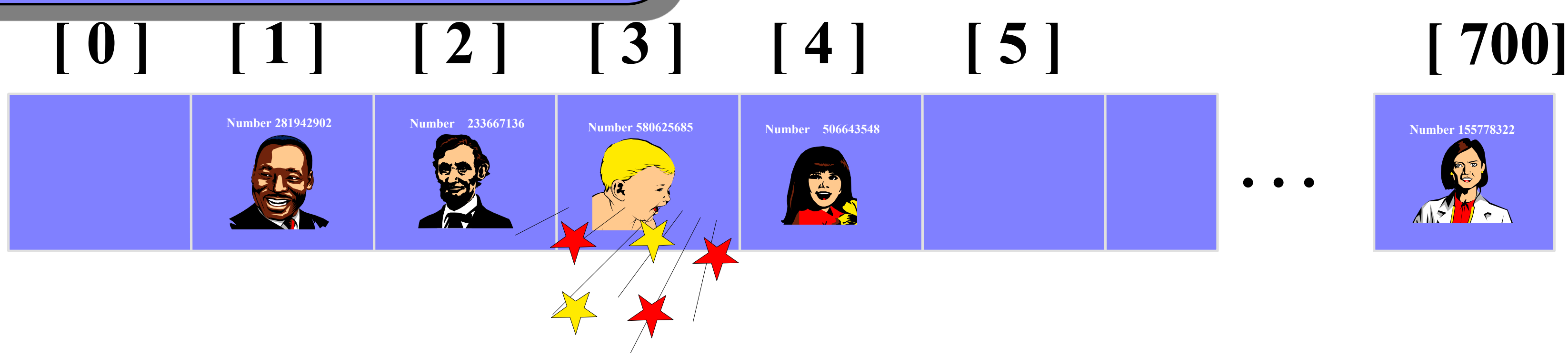




Collisions

- This is called a **collision**, because there is already another valid record at [2].

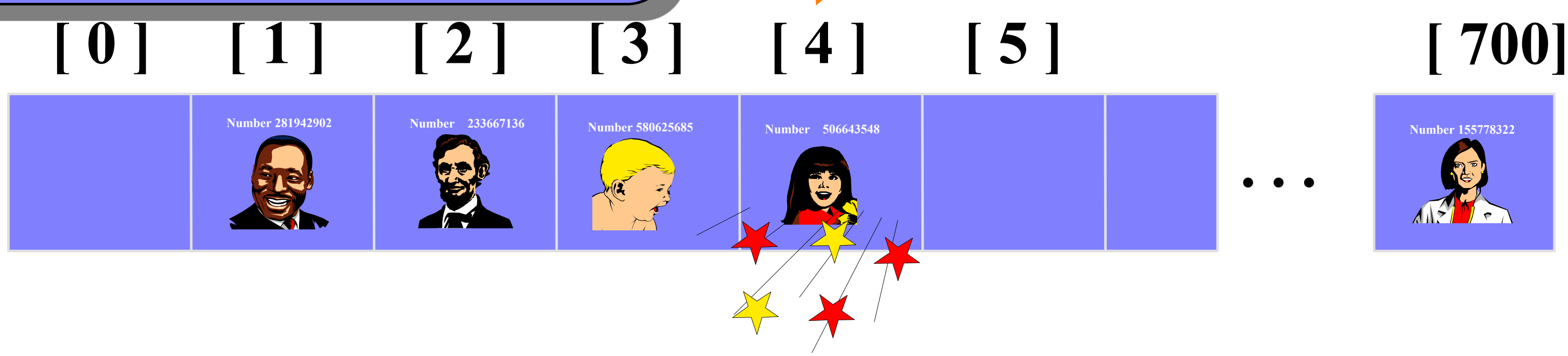
When a collision occurs,
move forward until you find an empty spot.



Collisions

- This is called a **collision**, because there is already another valid record at [2].

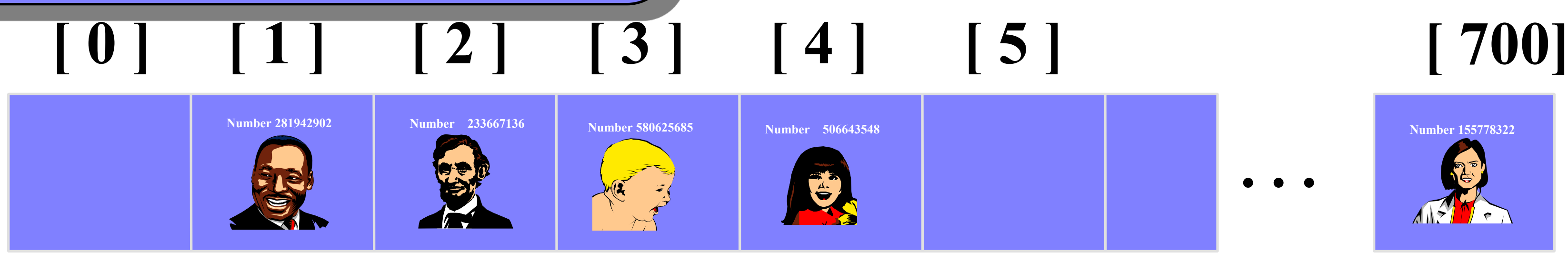
When a collision occurs,
move forward until you find an empty spot.



Collisions

- This is called a **collision**, because there is already another valid record at [2].

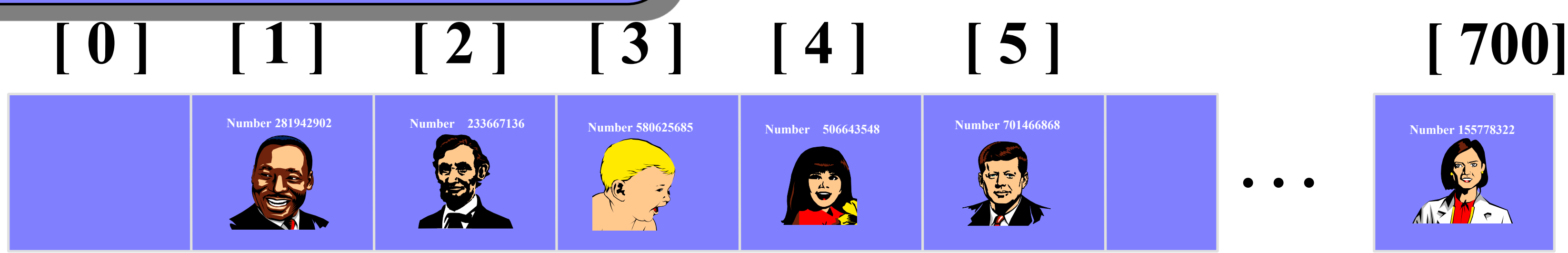
**When a collision occurs,
move forward until you find an empty spot.**



Collisions

- This is called a **collision**, because there is already another valid record at [2].

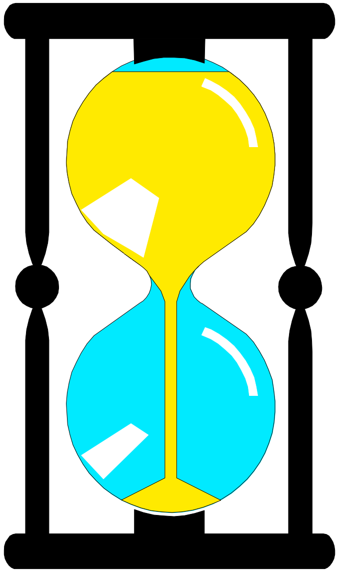
The new record goes in the empty spot.

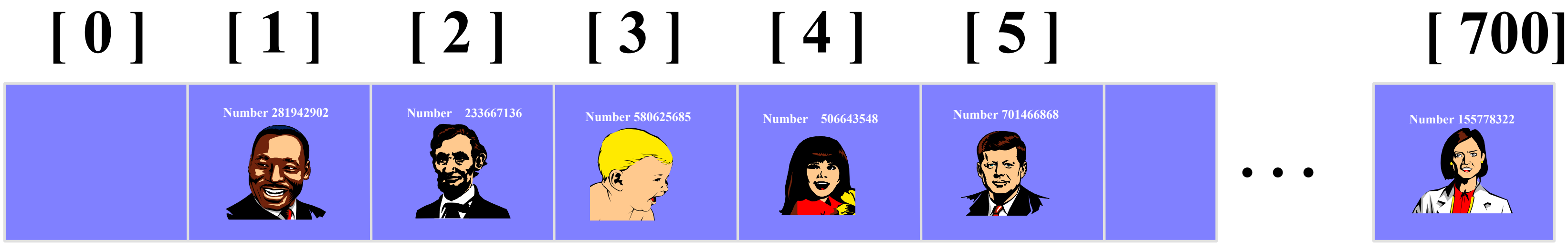


A small Quiz

At what index would you be placed in this table, if your NUMBER is 281942201

all slots from 6 to 699 are empty





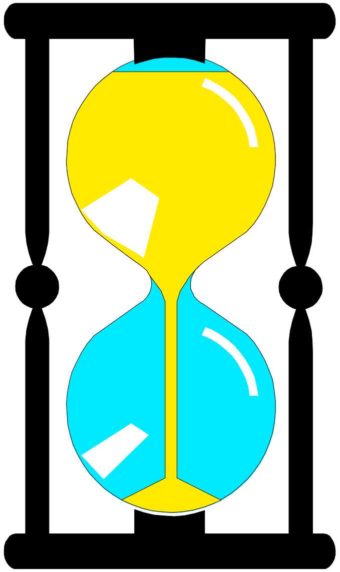
A small Quiz

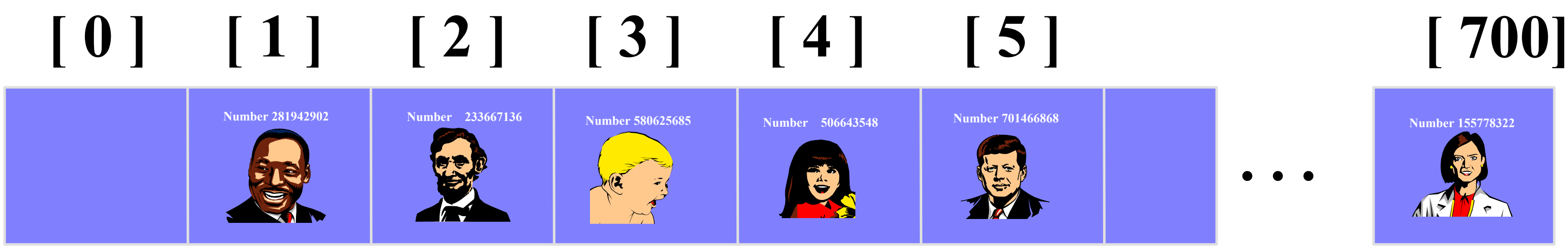
At what index would you be placed in this table, if your NUMBER is 281942201

all slots from 6 to 699 are empty

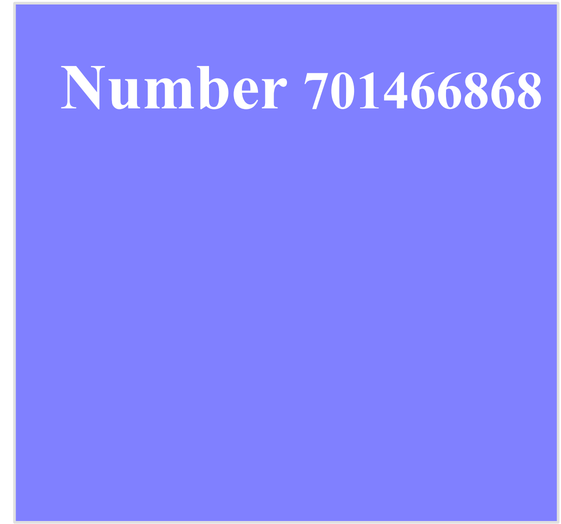
ANSWER = [6]

Explanation: 281942201 % 701 is [1], but due to collision, next available space is [6]

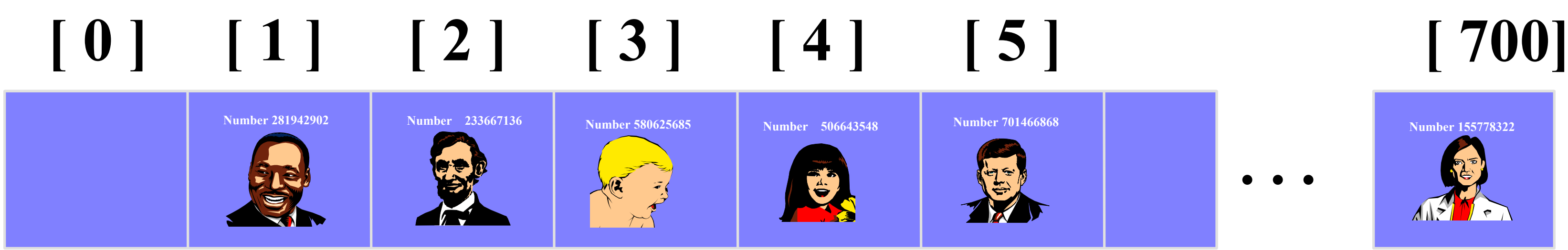




Searching for a Key

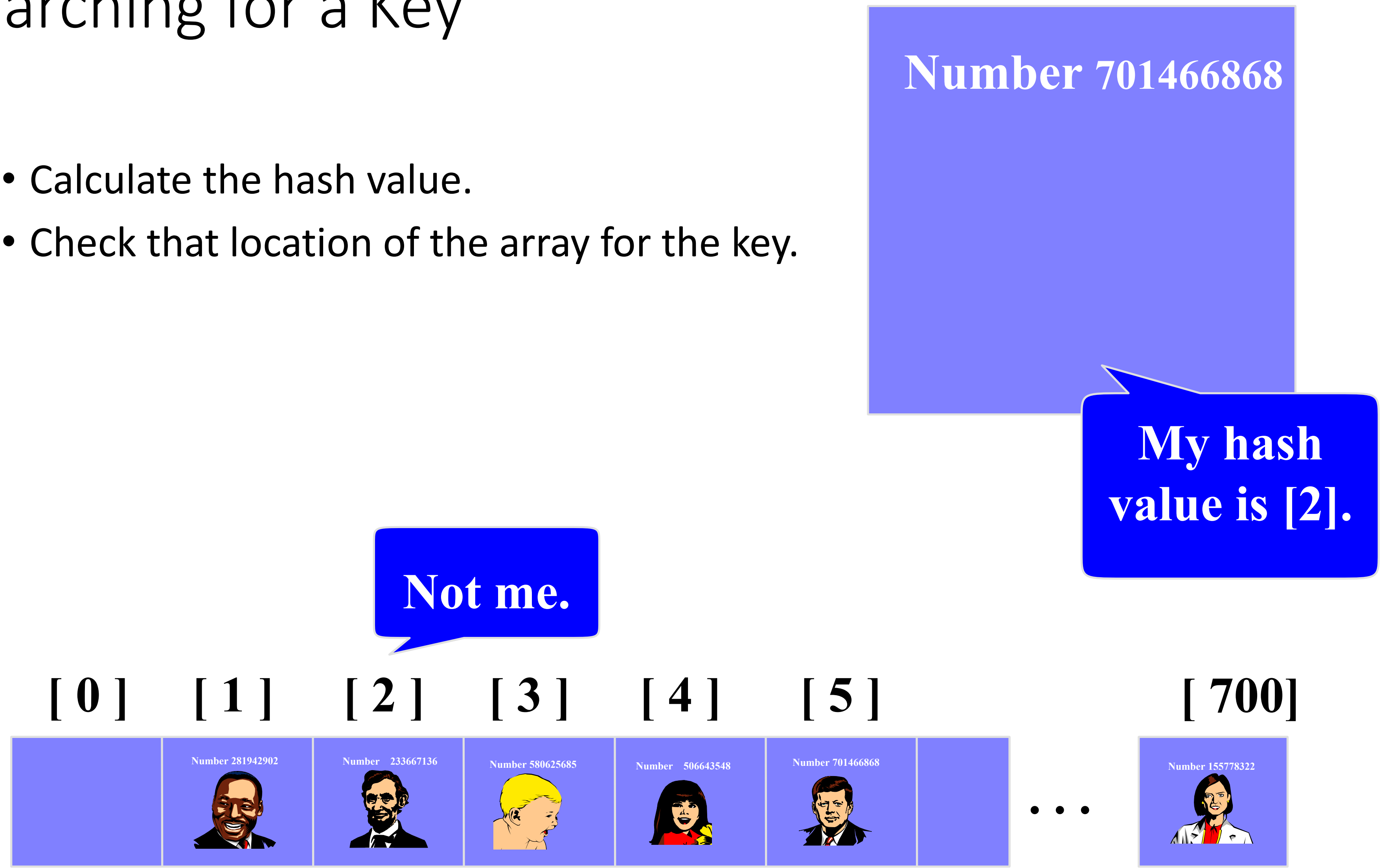


- The data that's attached to a key can be found fairly quickly.



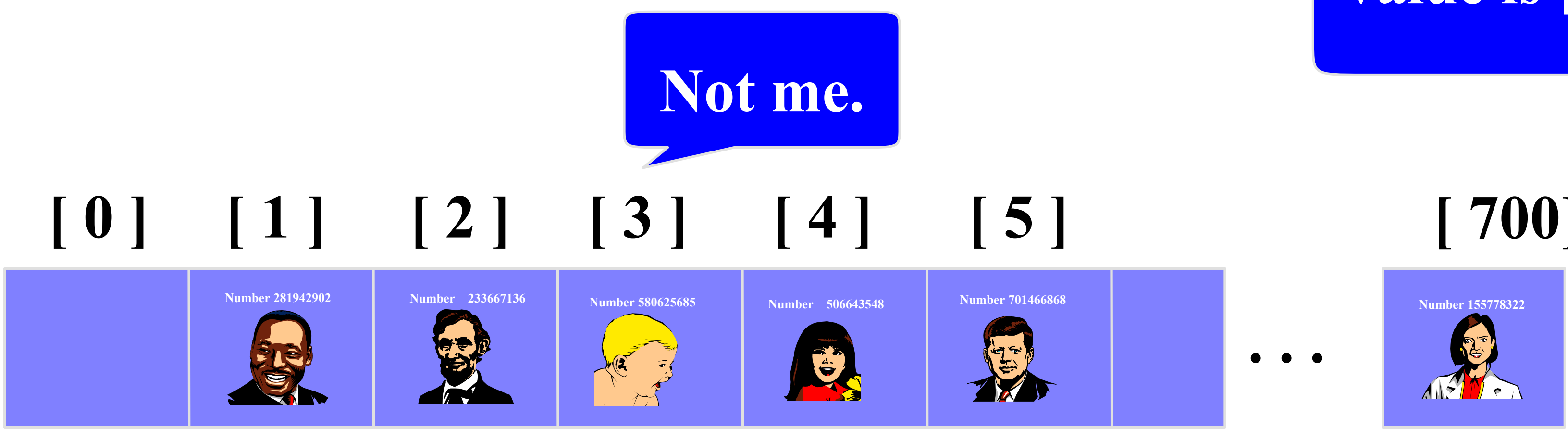
Searching for a Key

- Calculate the hash value.
- Check that location of the array for the key.



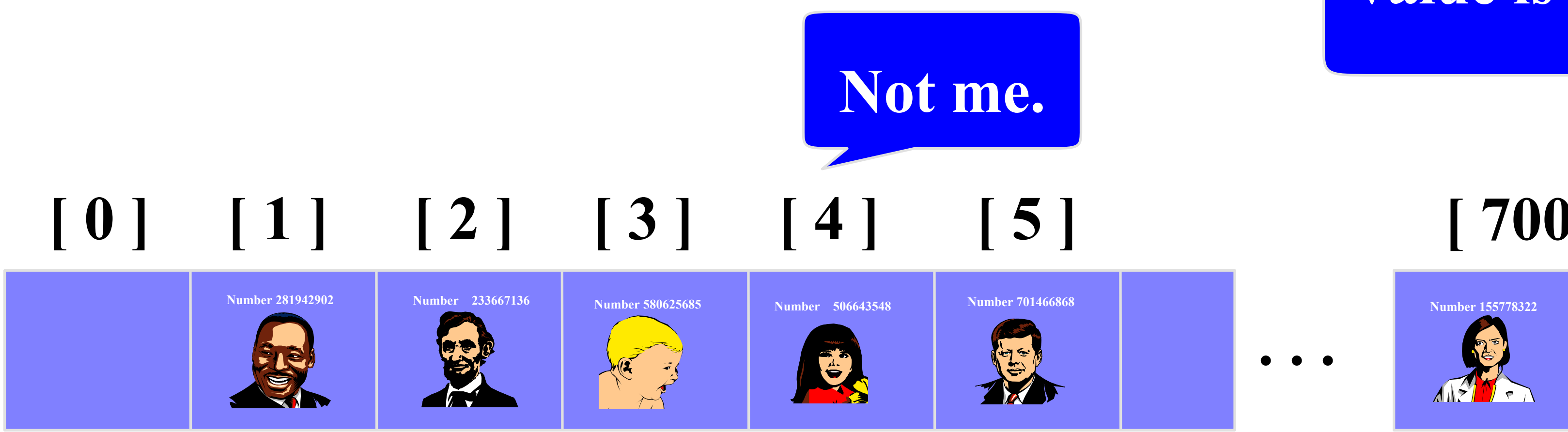
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.



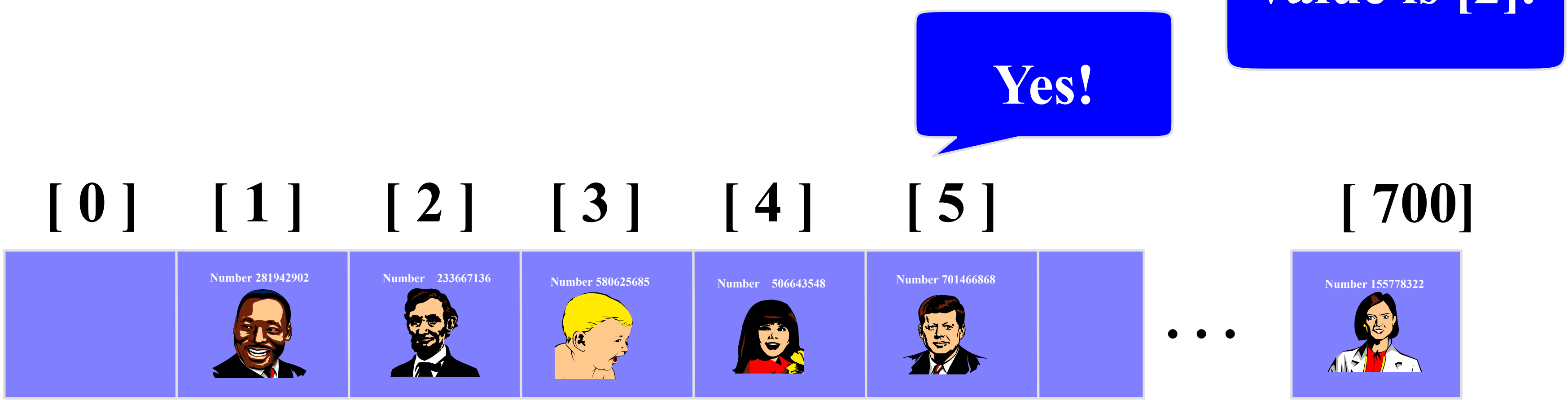
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot.



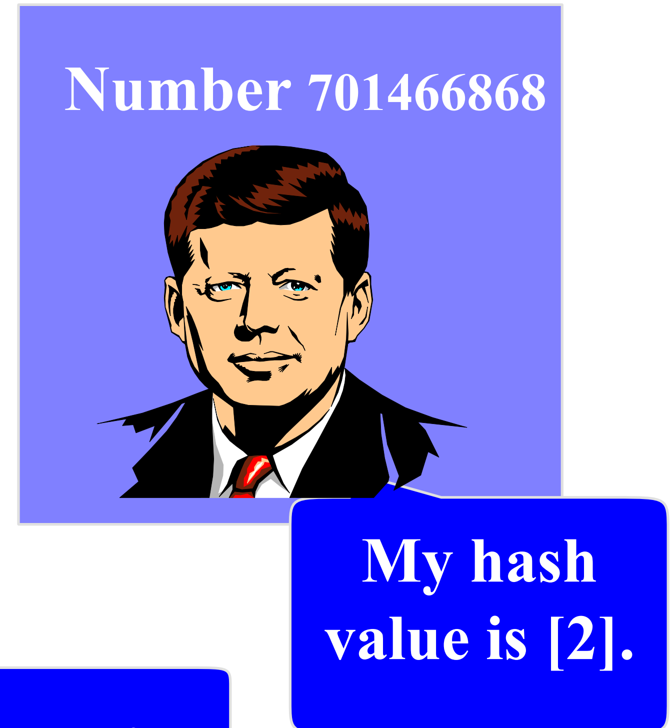
Searching for a Key

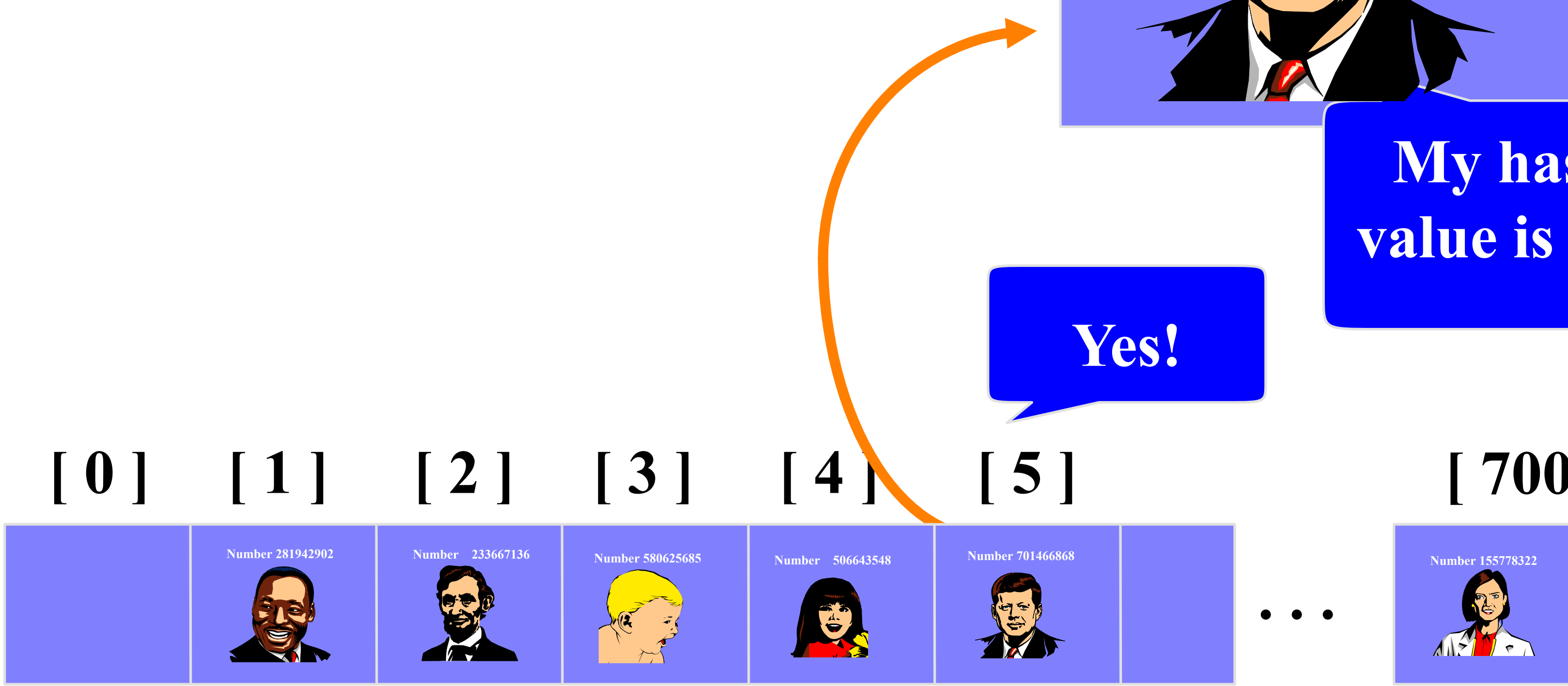
- Keep moving forward until you find the key, or you reach an empty spot.



Searching for a Key

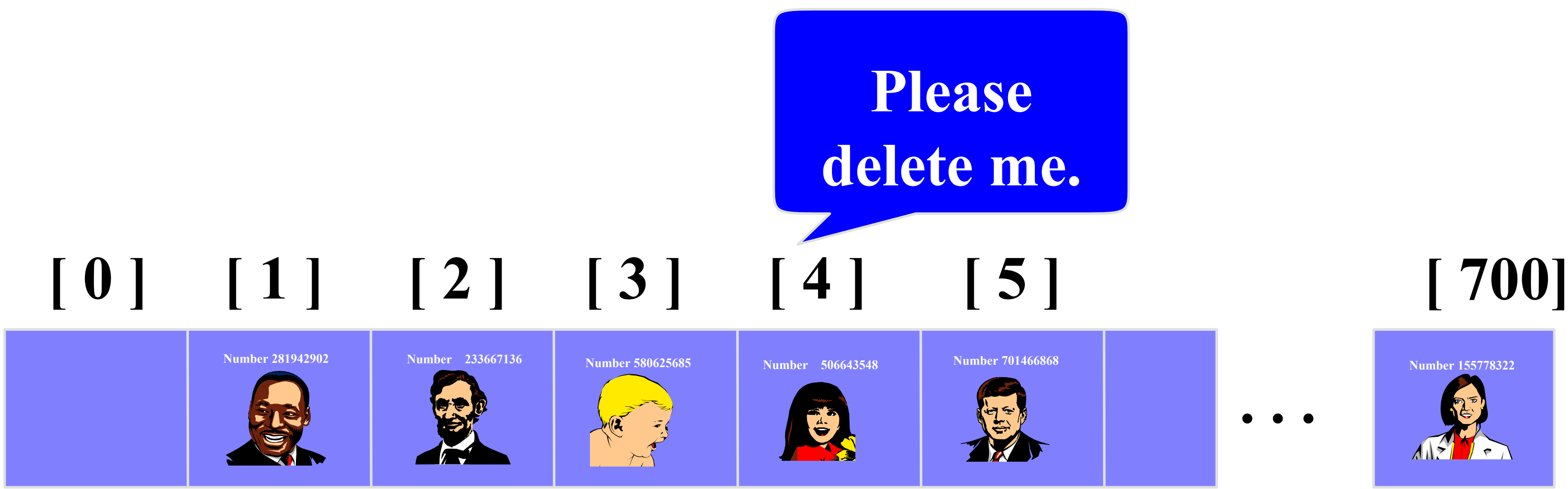
- When the item is found, the information can be copied to the necessary location.





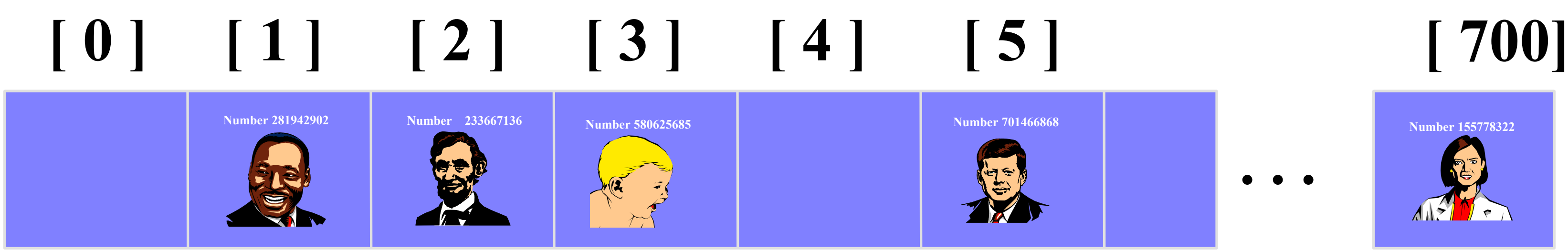
Deleting a Record

- Records may also be deleted from a hash table.



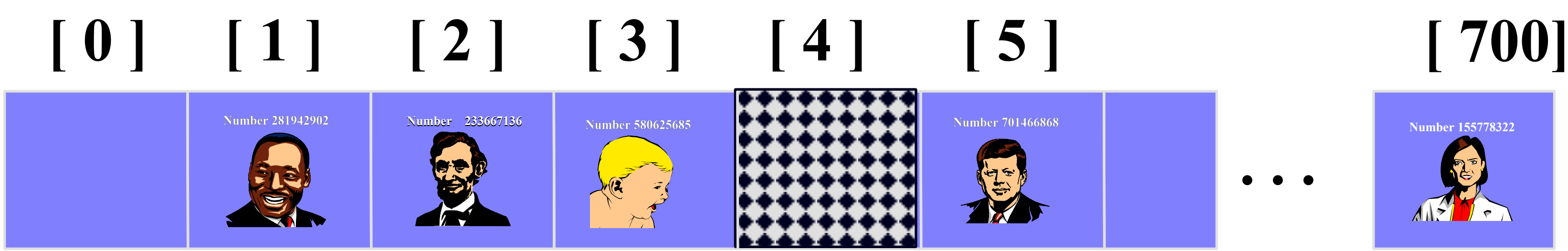
Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.



Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.
- The location must be marked in some special way so that a search can tell that the spot used to have something in it.



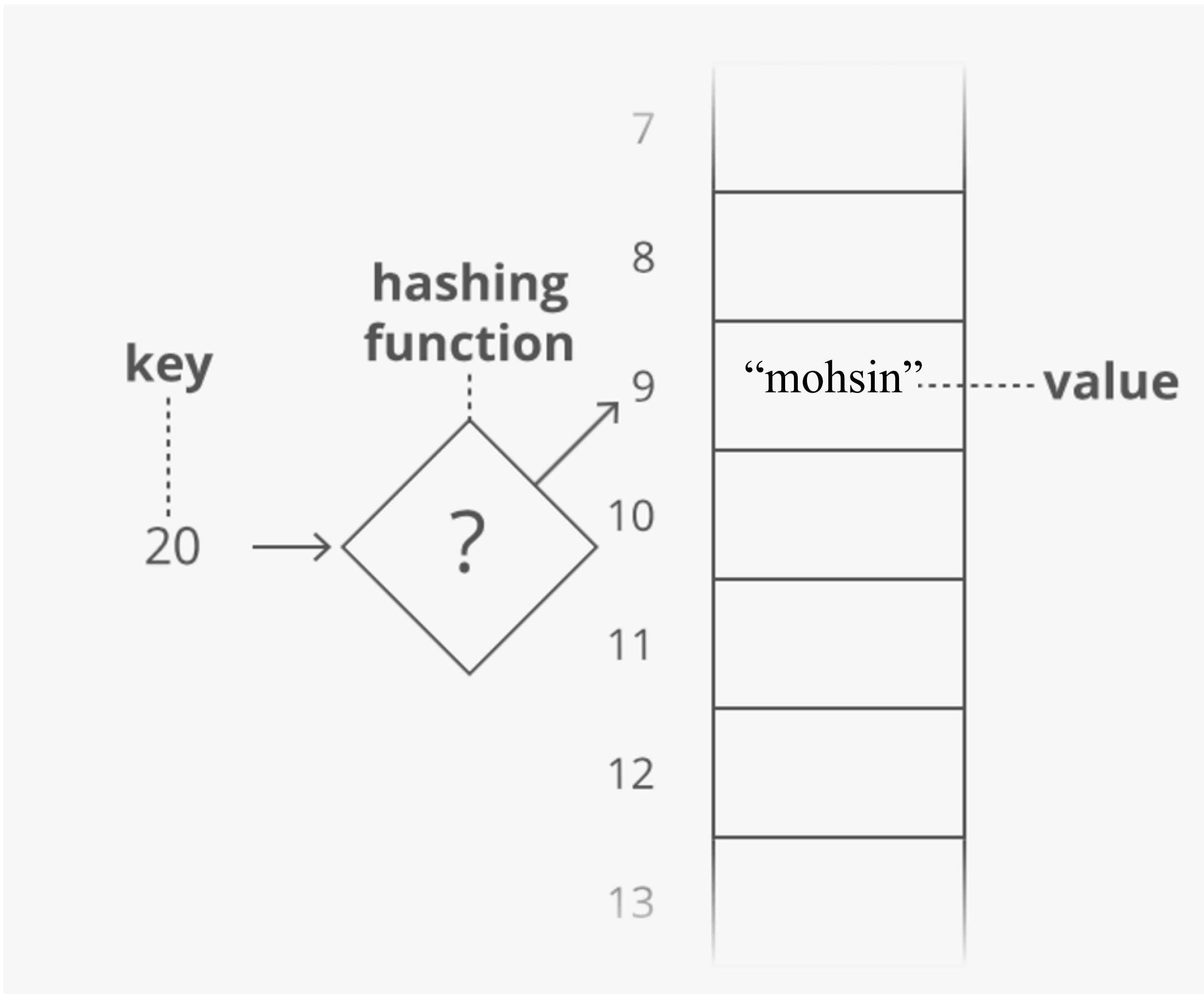
unordered_map<key, mappedValue>

STL

unordered_map<int, string>

Example:

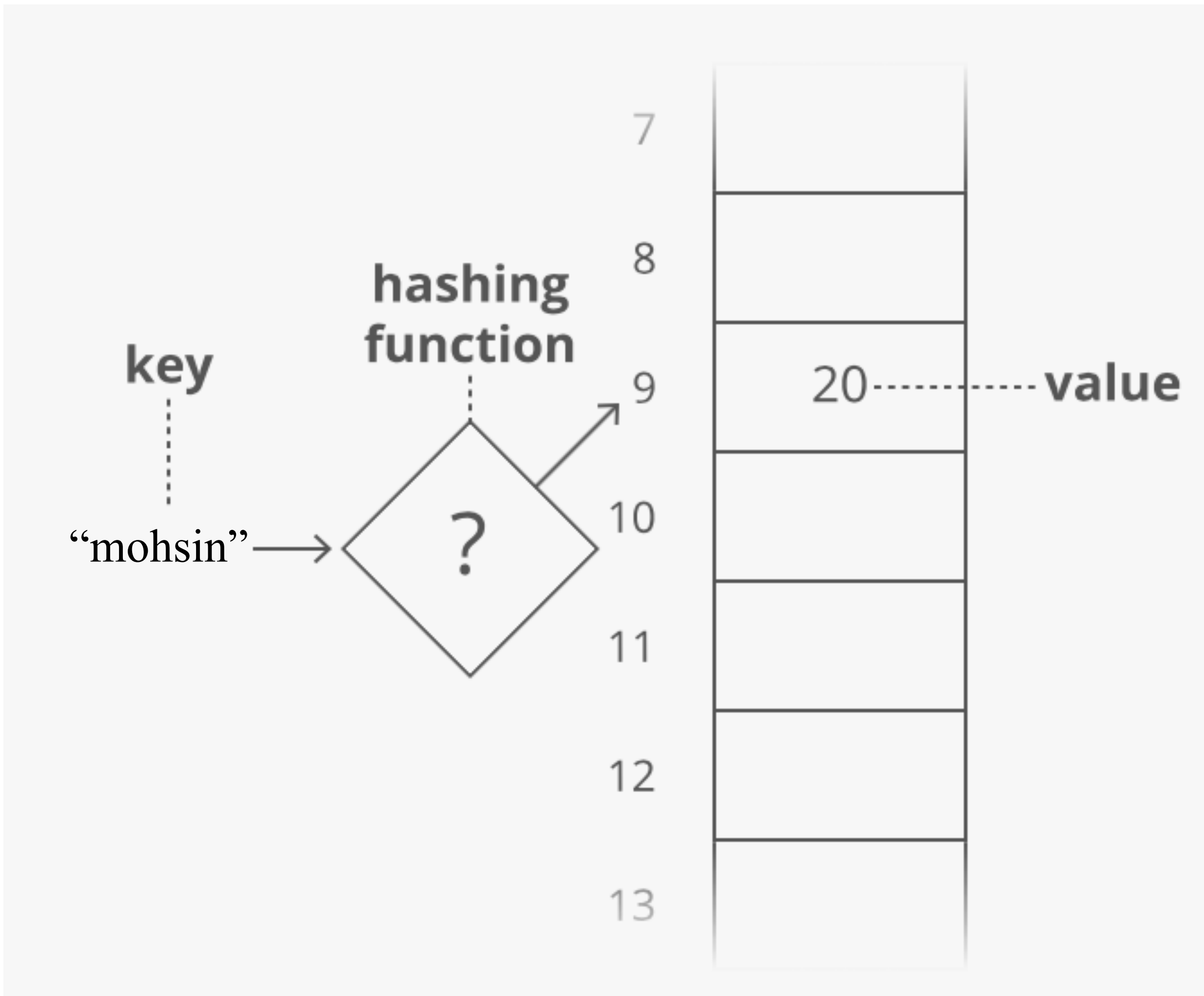
- key = 20
- mapped_value = “mohsin”



unordered_map<string, int>

Example:

- key = “mohsin”
- mapped_value = 20



Part 2

Hash Functions

Implementations So Far

	unsorted list	sorted array	Trees BST – average R-B – worst case
Search	O(n)	$O(\log_2 n)$	$O(\log_2 n)$

Properties of Good Hash Functions

- Must return an index number:
  0, 1, 2 …, [tablesize-1]
- Should be efficiently computable:
  O(1) time
- Should not waste space unnecessarily
  Load factor lambda λ = (no of keys / TableSize)
- Should minimize collisions

Integer Keys

- Hash(x) = x % TableSize
- Good idea to make TableSize *prime?* Why?

Suppose

data stored in hash table: 7160, 493, 60, 55, 321, 900, 810

tableSize = 10

data hashes to 0, 3, 0, 5, 1, 0, 0

tableSize = 11

data hashes to 10, 9, 5, 0, 2, 9, 7

Integer Keys

- Hash(x) = x % TableSize
- Good idea to make TableSize *prime?* Why?
- There is a high probability that collision will be avoided (it will not be eliminated however)

Collisions and their Resolution

- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!
- Two different methods for collision resolution:
  - **Separate Chaining:** Use a dictionary data structure (such as a linked list) to store multiple items that hash to the same slot
  - **Closed Hashing (or *probing*):** search for empty slots using a second function and store item in first empty slot that is found

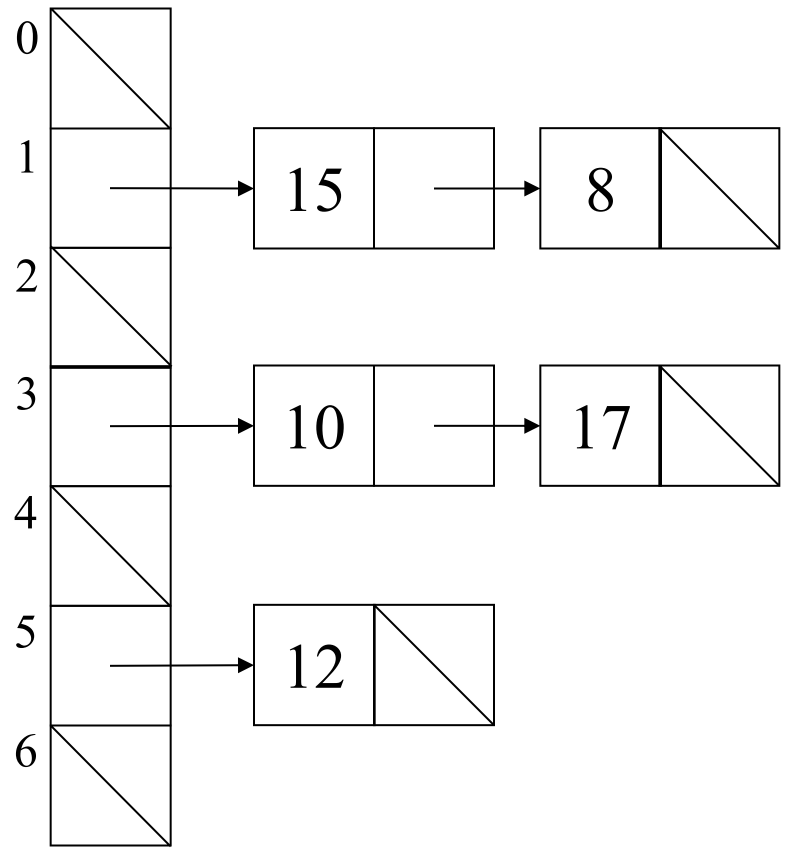
Terminology Alert

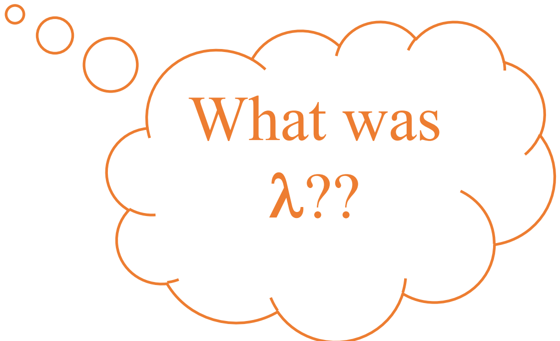
- Separate chaining = Open hashing
- Closed hashing = Open addressing

Hashing with Separate Chaining

- Common case is unordered linked list (chain)
- Properties
  - performance degrades with length of chains
  - **λ can be greater than 1**
- **Hash:**

  15, 10, 12, 8, 17





Collision Resolution by Closed Hashing

- $h_i(X) = (Hash(X) + F(i)) \bmod \mathit{TableSize}$
- F is the *collision resolution* function. Some possibilities:
  - Linear: $F(i) = i$
  - Quadratic: $F(i) = i^2$
  - Double Hashing: *2 hash functions*

Closed Hashing I: Linear Probing

- Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell
  - $h_i(X) = (\text{Hash}(X) + i) \bmod \textit{TableSize} \quad (i = 0, 1, 2, \ldots)$
  - Compute hash value and increment it until a free cell is found

Linear Probing Example

	insert(14)	insert(8)	insert(21)	insert(2)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2
0	14	14	14	14
1		8	8	8
2			21	12
3				2
4				
5				
6				

Drawbacks of Linear Probing

- Access time approaches O(N)
- Very prone to clusters
- Can have cases where table is empty except for a few clusters
  - Does not satisfy good hash function criterion of *distributing keys uniformly*

Closed Hashing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot – Increment by i^2 instead of i

- $h_i(X) = (\text{Hash}(X) + i^2) \ \% \ \textit{TableSize}$

  $h_0(X) = \text{Hash}(X) \ \% \ \text{TableSize}$

  $h_1(X) = (\text{Hash}(X) + 1) \ \% \ \text{TableSize}$

  $h_2(X) = (\text{Hash}(X) + 4) \ \% \ \text{TableSize}$

  $h_3(X) = (\text{Hash}(X) + 9) \ \% \ \text{TableSize}$

Quadratic Probing Example

	insert(14)	insert(8)	insert(21)	insert(2)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2
0	14	14	14	14
1		8	8	8
2				2
3				
4			21	21
5				
6				

Problem With Quadratic Probing

	insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
	14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2	7%7 = 0
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
					??

Problem With Quadratic Probing

	insert(14) $14\%7 = 0$	insert(8) $8\%7 = 1$	insert(21) $21\%7 = 0$	insert(2) $2\%7 = 2$	insert(7) $7\%7 = 0$
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					

Problem: Array Index Out of Bounds

Problem With Quadratic Probing

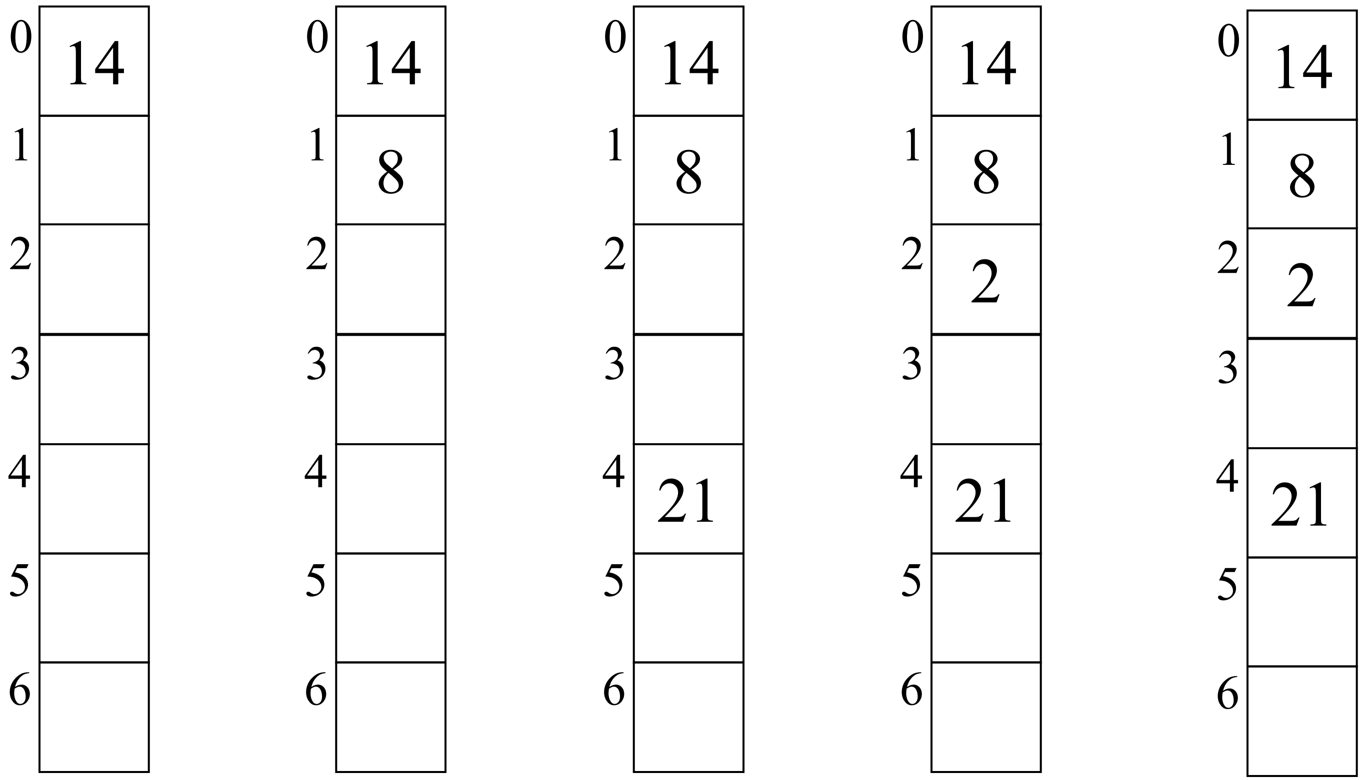
insert(14) — 14%7 = 0

insert(8) — 8%7 = 1

insert(21) — 21%7 =0

insert(2) — 2%7 = 2

insert(7) — 7%7 = 0



Solution: Huge ***TableSize*** required

Closed Hashing III: Double Hashing

- **Idea**: Spread out the search for an empty slot by using a second hash function
- Integer keys:
- $\text{Hash}_1(X) = X \bmod \mathit{TableSize}$

  $\text{Hash}_2(X) = R - (X \bmod R)$
  where R is a prime smaller than *TableSize*

- *Take R = 5 in our example*

Double Hashing Example

insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
14%7 = 0	8%7 = 1	21%7 =0	2%7 = 2	7%7 = 0
		5-(21%5)=4		5-(7%5)=3

0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					7
4			21	21	21
5					
6					